# Format Oracles on OpenPGP

Florian Maury, Jean-René Reinhard, Olivier Levillain, and Henri Gilbert*

ANSSI, France
{firstname.lastname}@ssi.gouv.fr

**Abstract.** The principle of padding oracle attacks has been known in the cryptography research community since 1998. It has been generalized to exploit any property of decrypted ciphertexts, either stemming from the encryption scheme, or the application data format. However, this attack principle is being leveraged time and again against proposed standards and real-world applications. This may be attributed to several factors, e.g., the backward compatibility with standards selecting oracle-prone mechanisms, the difficulty of safely implementing decryption operations, and the misuse of libraries by non cryptography-savvy developers. In this article, we present several format oracles discovered in applications and libraries implementing the OpenPGP message format, among which the popular GnuPG application. We show that, if the oracles they implement are made available to an adversary, e.g., by a front-end application, he can, by querying repeatedly these oracles, decrypt all OpenPGP symmetrically encrypted packets. The corresponding asymptotic query complexities range from 2 to $2^8$ oracle requests per plaintext byte to recover.

**Keywords:** GnuPG, Authenticated Encryption, Chosen Ciphertext Attacks, Padding Oracle, Format Oracle, Implementation.

## 1 Introduction

As defined in [4], a padding oracle attack is a particular type of side-channel attack where the attacker is assumed to have access to an oracle which returns `True` only when a chosen ciphertext corresponds to a correctly padded plaintext under a given scheme. Bleichenbacher [5] first applied this kind of attack to the PKCS#1 version v1.5 asymmetric encryption scheme. Vaudenay [15] showed that the same principle can be applied in the case of symmetric encryption when structured padding schemes are used. The "padding" terminology was introduced because the first attacks of this kind applied to specific padding schemes. They can be generalized to any format constraint on the plaintext providing redundancy, either imposed by the cryptographic scheme, or by the application using encryption, as illustrated by Klíma and Rosa on the PKCS#7

* This work was partially supported by the French National Research Agency through the BLOC project (contract ANR-11-INS-011)

format [10] and Mitchell [12]. We call this generalized form of oracle attacks *format oracle attacks*.

This type of attacks initially stems from the misconception that encryption mechanisms can provide a weak form of integrity through the following procedure: a format containing redundancy, e.g., fixed byte values, or linear relations, is applied to the plaintext before encryption. After decryption, it is checked whether the result satisfies the redundancy. Due to the malleability of some encryption schemes, and use of format properties that are satisfied with relatively high probability by random messages, this opens the way to chosen-ciphertext plaintext recovery attacks. Indeed, a format oracle leaks some information on the decryption of the submitted request. If all submitted requests are related to the same target ciphertext, its decryption may be obtained by aggregating the corresponding information leakage.

Format oracle answers come in different flavours. They all rely on a variation of the behaviour of the decryption procedure related to some property of the decrypted value: specific byte values expected at some positions or high-level consistency constraints for example. The most explicit forms of information leakage are characteristic error messages.Format oracles can also be obtained by exploring logged information. Finally, more implicit oracles, relying on timing leaks, memory caching strategies, and other side-channels, are also possible. Even though the principle of padding, and format, oracles has been known for over 15 years, numerous publications [1, 7, 14, 4, 2, 13] attest that they are quite pervasive, and may continue to be instantiated in modern applications.

A general countermeasure against these attacks consists in checking the integrity of ciphertexts before performing any decryption, thus eliminating any chosen-ciphertext attack possibility by construction. Unfortunately, due to backward compatibility issues, many standards still do not support proper authenticated encryption. Moreover, this may require a two-pass authenticated decryption that may be impractical when large streams of data are processed. As a consequence, a less satisfying fallback solution has been adopted in several contexts: ensuring implementations do not instantiate format oracles in order to avoid the exploitation of these attacks. This sometimes leads to convoluted implementations, since one has to ensure that no side-channel leaks information. Another concern is the misuse of cryptographic toolkits and libraries. Such software is developed by programmers proficient in cryptography. They strive to make their implementation resistant against state-of-the-art attacks, by selecting robust cryptography, and by avoiding side-channel leakage. Yet, most programmers using cryptographic libraries are not expert cryptographic security evaluators. They can legitimately expect them to behave as secure modules, unless explicetely advised otherwise. Therefore, if sufficient warnings are not made, they may incorrectly perceive some of their outputs, e.g., sensitive error messages, as innocuous.

OpenPGP [9] is a message format used to preserve privacy by providing encryption. It is notably implemented in the popular GnuPG toolkit. Recently,

two JavaScript libraries, OpenPGP.js and Google-backed End-to-End, have been released.

Previous attacks against OpenPGP implementations leveraged the malleability of the encryption mode used in OpenPGP to recover plaintexts. A first attack [8], by Jallad, Katz and Schneier, achieves complete decryption of ciphertexts but requires access to a decryption oracle, that may be implemented by an inadvertent user transmitting random-looking decryption results to the adversary. A second attack [11], by Mister and Zuccherato, takes advantage of a less powerful oracle related to the CFB-mode variation used by OpenPGP to detect the use of an erroneous decryption key and enables to recover two bytes from every ciphertext block. This second attack is an example of a padding oracle attack. Some mitigation measures have been adopted against these attacks, like removing the CFB-mode oracle when decryption relies on asymmetric cryptography, or introducing encryption with integrity. However some implementations still leak in certain use cases the information exploited by these attacks: the security ultimately relies on the careful use of the application.

The main contribution of this article is the identification of several new format oracles in OpenPGP implementations. We show that many OpenPGP applications and libraries, e.g., GnuPG, OpenPGP.js, and End-to-End, actually leak sensitive information in error messages raised during the decryption of OpenPGP encrypted messages.If these error messages are mishandled, e.g., by a front-end application, the identified oracles can be leveraged to fully decrypt any encrypted message, thus demonstrating that the error messages are not innocuous with regards to confidentiality. This leads us to believe that the handling of errors, e.g., decryption errors, is a part of the API of cryptographic libraries that should receive more attention. To minimize the risk of implementation errors, cryptographic library providers should prevent any unnecessary leakage of information, and clearly identify the elements of the API that are sensitive.

Similarly to previously published padding oracle attacks, these attacks are chosen-ciphertext attacks requiring interactions with a legitimate recipient of the target message. The complexity of these format oracle attacks ranges from 2 to $2^8$ queries to the format oracle per byte to decrypt, according to the leveraged oracle. We implemented these attacks against GnuPG and experimentally confirmed their complexity.

We reported our findings to the developers of the mentioned OpenPGP implementations. They took them into account by patching their implementations to remove some possible oracles (cf Section 5 for details).

In Section 2, we give an overview of the OpenPGP message format and its (authenticated) encryption mechanism. Section 3 presents format oracles, which are implemented by GnuPG, OpenPGP.js, and End-to-End, when they are viewed as libraries, and Section 4 how these format oracles can be leveraged to decrypt ciphertexts. In Section 5, we discuss countermeasures to thwart them.

*Notations.* We denote $E$ a block cipher and $n$ its blocksize expressed in bytes. $E_K$ denotes encryption under key $K$.

Let $||$ denote the concatenation. Let $P$ be a non-empty message $P \in (\{0,1\}^8)^*$. Let $|P|$ be its byte length. It can be decomposed into a sequence of blocks $P_1||P_2||\ldots||P_m$, where $P_i$ is an $n$-byte block for $i < m$, and $P_m$ is a non-empty, possibly incomplete, block.[1] Let $\|P\| = m$ denote the number of blocks of $P$ in this decomposition. Furthermore, for $j \in [1, |P|]$, let $P[j]$ be the $j$-th byte of message $P$. $P_i[j]$ is the $j$-th byte of the $i$-th block of $P$. For $j \in [1, |P|]$, $P[-j]$ is the $j$-th byte from the end: $P[-j] = P[|P| + 1 - j]$. Let $\overline{P} = P[-2]||P[-1]$ be the concatenation of the last two bytes of $P$.

Explicit byte values are given in hexadecimal form, e.g., `0xD3`. $\texttt{0x00}_u$ denotes the concatenation of $u$ zero bytes.

## 2 OpenPGP Format Description

### 2.1 Packet Structure of OpenPGP Message Format

**Overview.** All values (data, keys, etc.) considered by OpenPGP are structured and processed in *packets*. Well-formed OpenPGP messages follow a grammar described in [9, section 11.3], which specifies a recursive composition of packets and OpenPGP messages: an OpenPGP message is a concatenation of packets, some of which may contain a processed form of an OpenPGP message. Each packet is a sequence of bytes, with a (tag, length, value) structure.[2] The first byte, called the *tag*, encodes the type of information that the packet contains. The length field encodes the length of the value field. The value contains the payload of the packet, and its structure depends on the considered packet type.

**Data Packet Structures.** All user data is found either in literal, compressed, or encrypted packets. Encrypted packets come in two flavours, one providing only confidentiality, another providing both confidentiality and integrity protection.

*Literal Packets.* We denote $T_\ell$ the one-byte tag value of literal packets. The literal packet $\texttt{LitPacket}(D)$ stores data $D$ in an unprocessed way, preceded by a header containing some metadata, e.g., a file name or a date.

*Compressed Packets.* We denote $T_c$ the one-byte tag value of compressed packets. The compressed packet $\texttt{CompPacket}^{T_a}(Z)$ stores the value $Z$ resulting from the compression under algorithm $T_a$ of an OpenPGP message. The payload of compressed packets contains a byte $T_a$, encoding the compression algorithm, followed by the compressed value $Z$.

*Encrypted Packets.* Let $T_e$ be the one-byte tag value of encrypted data packets. The encrypted packet $\texttt{EncPacket}_K^E(C)$ without integrity protection stores the ciphertext resulting from an encryption. The payload is simply the ciphertext $C$ resulting from the encryption of an OpenPGP message using block cipher $E$ with key $K$. The encryption procedure is detailed in Section 2.2.

[1] We abusively also refer to this part of the decomposition as a block.

[2] This is an approximation, since the new length format introduced in the specification [9] supports partial length values, but this does not affect our attacks.

*Encrypted Integrity Protected Packets.* We denote $T_E$ the one-byte tag value of encrypted integrity protected data packets. The encrypted integrity protected packet $\texttt{EncIntPacket}_K^E(C)$ stores data after integrity protection and encryption steps, detailed in Section 2.2. The payload of the packet contains a version byte, set to 1, followed by the ciphertext $C$ resulting from the encryption of an OpenPGP message using block cipher $E$ with key $K$.

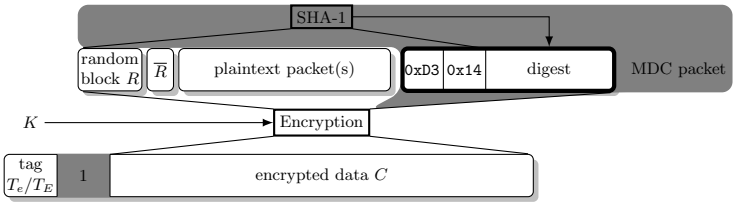


Fig. 1: Encrypted packets format and encryption procedures. The elements on gray background only appear in encrypted integrity protected packets.

**Key Packets.** The generation/decryption of the ciphertext contained in an encrypted packet, with or without integrity protection, involves a secret key $K$ that is securely stored in a key packet $\texttt{KeyPacket}_\text{A}(K, E)$, that is transmitted along with the encrypted packet. This key packet also encodes the block cipher primitive $E$ selected by the sender. Only recipient A can extract the key from the key packet, because he either shares a passphrase with the sender, or he owns a given private key. The latter is the general use case, and is a form of hybrid encryption, where asymmetric encryption protects a message key used to symmetrically encrypt data. In the following, we assume that the legitimate recipient unlocks the message key, and focus on the symmetric encryption mechanism.

### 2.2 Encryption Procedures

Encryption with integrity protection uses a block cipher in CFB mode with an all-zero IV. Initial encryption randomization is obtained by prepending to the plaintext packet(s) a block $R$ of $n$ random bytes. The last two bytes of the random block $\overline{R} = R[-2]||R[-1]$ are repeated as the first two bytes of the second block. This redundancy provides an early way to detect the use of a wrong key derived for example from an erroneous passphrase. This test was used in [11] to recover 16 bits of plaintext per block.

A suffix starting with two fixed byte values, $\texttt{0xD3}||\texttt{0x14}$ is appended. These represent the header (tag and length) of an OpenPGP Modification Detection Code (MDC) packet. Then, a SHA-1 digest is computed over the concatenation of the prefix, the plaintext packet(s), and the header of the MDC packet. The resulting 20 bytes are the payload of the MDC packet, which are further appended

in order to obtain the input $P$ of the encryption. A graphical representation of the encoding of $P$ can be found in Figure 1. The CFB chaining equation is given by the following formula[3]:

$$C_i = E_K(C_{i-1}) \oplus P_i \tag{1}$$

We comment briefly on the security of encryption with integrity protection in OpenPGP in Appendix A.

Encryption without integrity protection presents only slight variations. First, no suffix is appended after the plaintext. Second, the state of the CFB encryption function is resynchronized after the encryption of the prefix. That is to say, the first $n + 2$ bytes are encrypted using CFB mode, then the CFB state is set to the last $n$ bytes of the current ciphertext, i.e., starting from the third byte, and the encryption of the plaintext in CFB mode resumes from this point.

## 3 Format Oracles

A format oracle describes a side-channel leaking information on the decrypted values corresponding to given ciphertexts. Format oracles generalize padding oracles in two directions. Firstly, the test performed by the oracle is not restricted to the padding of the plaintext, but can also take into account any constraint of the (application-related) format of the plaintext. Secondly, the format oracles may leak plaintext information faster than classical padding oracles. We say a format oracle is *boolean* when its output is simply the result of some format verification on the decrypted ciphertext, and *leaky* if it provides additional information.

As we shall see in the rest of this section, OpenPGP libraries leak through their error messages partial information on the decrypted values corresponding to submitted ciphertexts. For example, GnuPG generally emits non-fatal errors on the standard error stream `stderr` if the result of decryption presents format inconsistencies. Among these messages, it is worthwhile to distinguish so-called *status messages*, that are specifically intended to provide information to applications using GnuPG as a backend. Thus, any leakage through status messages is particularly worrisome.

Should the mentioned error messages be made available to an adversary by an inadvertent application, a format oracle would be instantiated. In particular, JavaScript libraries are expected to run in possibly hostile environment, where exceptions and error messages should be sanitized.

It is difficult to formalize completely the oracle definitions, because they rely on the format of OpenPGP messages. We shall see in the Section 4 that for specially crafted ciphertexts, most of these oracles enable to test whether a decrypted ciphertext byte takes a value in a specific set, possibly a singleton.

[3] By convention $C_0$ is the all-zero IV, and is not transmitted. Note also that in case the last $P_i$ is shorter than a block, the value of $E_K(C_{i-1})$ is truncated accordingly.

**The *Invalid Identifier* Oracles.** An OpenPGP packet contains several constrained values. For example, the RFC specifies that the packet tag is a one-byte value with its most significant bit (MSB) set to 1. Such constrained identifiers are pervasive in the OpenPGP specification: packet tags, compression methods, encryption algorithms, etc.

**Definition 1.** *An* invalid identifier *oracle is a leaky oracle taking as input a key packet* $\texttt{KeyPacket}_{\mathrm{A}}(K, E)$ *containing a key* $K$ *for an algorithm* $E$*, and a symmetrically encrypted packet containing a ciphertext* $C$*. It tests whether, when parsing the packets in the decrypted ciphertext, all bytes interpreted as some type of identifiers respect the associated constraints. Furthermore, it leaks the values of the bytes that do not satisfy this format.*

The GnuPG error messages `invalid packet (ctb=XX)`, where `XX` is an offending packet tag, with MSB equal to 0, providean *invalid tag* oracle. The OpenPGP.js exception messages `Compression algorithm XX not implemented` provide an *invalid compression method* oracle leaking the offending compression method `XX`. For End-to-End, the `Unsupported id: XX` exception messages provide an *invalid algorithm identifier* oracle.

**The *Double Literal* Oracle.** The OpenPGP specification [9] states that only a single literal data packet may be found in any OpenPGP message. OpenPGP implementations may check that only one such packet exists in a message or otherwise emit a specific error message. This leads us to consider the following format oracle:

**Definition 2.** *The* double literal *oracle is a boolean format oracle that takes the same input as the* invalid identifier *oracle. It tests whether the tags of any two consecutive OpenPGP packets in the decrypted ciphertext are both literal packet tags.*

The GnuPG `WARNING: multiple plaintexts seen` error messages, and the `proc_pkt.plaintext 89_BAD_DATA` status messages[4] provide a *double literal* oracle. OpenPGP.js provides this oracle as well.

**The *MDC Packet Header* Oracle.** A potential format oracle can be found in the encryption with integrity protection mechanism of OpenPGP. As described in Section 2, the input of the encryption function during the encryption with integrity protection process is infused with some format so that the plaintext is the concatenation of an OpenPGP message followed by an MDC packet. Furthermore, OpenPGP mandates the use of SHA-1 as the hash function for MDC packets during encryption, and defines the header of these packets (tag and length). Thus, for the decryption of a legitimate ciphertext, the 22nd and 21st bytes counting from the end after decryption have prescribed values. This leads us to consider the following format oracle:

[4] These status message are emitted unless they are explicitely inhibited by the caller, through setting the flag `--allow-multiple-messages`.

**Definition 3.** *The* MDC packet header *oracle is a boolean format oracle that takes as input a key packet* $\texttt{KeyPacket}_{\text{A}}(K, E)$ *containing a key* $K$ *for an algorithm* $E$*, and* $\texttt{EncIntPacket}_K^E(C)$*, a symmetrically encrypted integrity protected packet. Denoting* $P$ *the decrypted ciphertext, the oracle tests whether* $P[-22] = \texttt{0xD3}$ *and* $P[-21] = \texttt{0x14}$*.*

GnuPG `mdc_packet with invalid encoding` error messages provide an MDC packet header oracle, as well as its `DECRYPTION FAILED` status messages combined with the absence of a `BAD MDC` status messages. `Modification Detection Code not properly formatted` error messages, that can be returned by a low-level function of the End-to-End library, provide the same oracle.

The definition of the MDC packet header oracle is a slight simplification of the behaviour that can be found in OpenPGP libraries. More details are given in Appendix B.

## 4 Plaintext Recovery Attacks

### 4.1 Overview of Format Oracle Attacks against OpenPGP

In the following, we consider an encrypted packet whose payload contains the target ciphertext value $C^*$, and a corresponding key packet $\texttt{KeyPacket}_{\text{A}}(K, E)$. Note that the attacker does not have access to the key $K$ that protects the target encrypted packet. Only user A can recover $K$ from the key packet, using a passphrase or a private key. The objective of the attacker is to recover the decryption $P^*$ of $C^*$ by leveraging a format oracle implemented by user A. The attacker performs several requests to the oracle, with specially crafted, format oracle specific, ciphertexts $C = C(B, u, a)$, were $B$ is the target block, $u$ is the target position in the block, and $a$ is the tested value. The oracle answers leak information allowing to decrypt ciphertext $C^*$ step by step.

OpenPGP symmetric encryption relies on variants of the CFB mode of operation. Let us notice that in order to decrypt a CFB encrypted message, it is sufficient to be able to recover the encrypted value of any block. Applying this recovery procedure on block $C_i^*$, we get $E_K(C_i^*)$ which can be used to decrypt $C_{i+1}^*$ through Equation 1: $P_{i+1}^* = C_{i+1}^* \oplus E_K(C_i^*)$.

The attacks presented enables to attack any type of encrypted data, be it encrypted packets with/without integrity protection, or symmetrically encrypted session key packets. More details are given in Appendix C.

A general attack overview is given in Algorithm 1. The attack complexities are expressed in the maximal number of queries to the format oracle. The corresponding average complexity is half the maximal complexity. We now describe, for the format oracles described in Section 3 the structure of the submitted ciphertexts $C(B, u, a)$. The presented query ciphertexts only contain the payload specific to the studied oracle. If the decryption function enforces other constraints, e.g., checks the initial redundancy of OpenPGP plaintexts, the query ciphertexts can be tweaked, as described in Appendix B.

**Algorithm 1** Transforming a format oracle into an attack: overview

```
for all ciphertext blocks C_i^* do
  for all (byte) positions 1 ≤ u ≤ n in the ciphertext block do
    for all possible (byte) values a do
      Submit ciphertext C(C_i^*, u, a) to the oracle
      if oracle returns True then
        deduce the value of E_K(C_i^*) at position u from a
```

### 4.2 Plaintext Recovery using Tag Oracles

First we describe how to leverage an invalid tag oracle, that is a specific case of an invalid identifier oracle, or a double literal oracle. The same principle can be applied to leverage any invalid identifier oracle. In order to recover $E_K(B)[u]$, we consider ciphertexts with the following format:

$$C(B, u, a) = T_\ell \oplus \alpha || U \oplus \beta || \texttt{0x00}_{n-2} || B || \texttt{0x00}_{u-1} || a,$$

where $U$ is the byte whose value is $2n + u - 3$, and $\alpha$ and $\beta$ are the bytes used to decrypt the first two bytes of ciphertext. We can assume that these two bytes can be predicted by an attacker from $C^*$ (cf Appendix B).

These ciphertexts are built so that the corresponding plaintexts contain a first literal packet of length $U = 2n + u - 3$. The zero paddings ensure that $B$ starts at a block boundary, and that the tag of the second packet is located at position $u$ of the block following $B$. For the double literal oracle (resp. for the invalid packet tag oracle), we let $a$ (resp. the MSB of $a$) take all possible values. After decryption, this lets the tag (resp. the MSB of the tag) of the second packet take all possible values. The format oracle returns `True` if $E_K(B)[u] \oplus a$ is a valid tag (resp. it leaks $E_K(B)[u] \oplus a$ if its MSB is 0). We give a graphical representation of this procedure in Figure 2. The maximal number of requests to recover this byte is approximatively $2^6$ (resp. 2). Only about $2^6$ requests are needed instead of $2^8$, because there are 5 possible tag values (cf [9]).

### 4.3 Plaintext Recovery using the MDC Packet Header Oracle

**Basic Recovery of the Encrypted Value of a Block.** To recover $E_K(B)$, we consider the ciphertexts with the following format:

$$C(B, u, (a, b)) = B || \texttt{0x00}_{u-1} || a || b || \texttt{0x00}_{20}.$$

$B$ is block aligned, $1 \leq u \leq n - 2$ pads the ciphertext so that the target bytes are located at position $u$ and $u + 1$ of the following block, and the final 20 zeros ensures that $a$ and $b$ are considered as the MDC header.

*Recovering $E_K(B)[u]$ and $E_K(B)[u + 1]$.* In order to recover two consecutive bytes of $E_K(B)$, we let $a$ and $b$ take all possible $2^{16}$ values. This lets the bytes located at the MDC packet header position after decryption take all $2^{16}$ possible

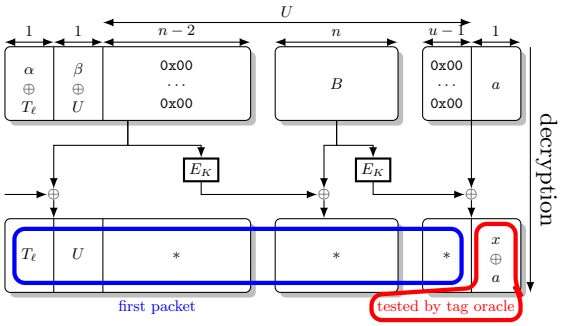


Fig. 2: Recovery procedure of $x = E_K(B)[u]$, using a tag oracle.

values. Thus the format oracle returns `True` for a unique pair $a', b'$, and we have, through Equation 1: $E_K(B)[u] = \texttt{0xD3} \oplus a', E_K(B)[u+1] = \texttt{0x14} \oplus b'$. We give a graphical representation of a special case of this procedure in Figure 3. The maximal number of requests to recover these bytes is $2^{16}$.

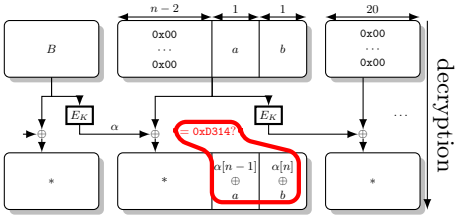


Fig. 3: Recovery of the last two bytes of $E_K(B)$.

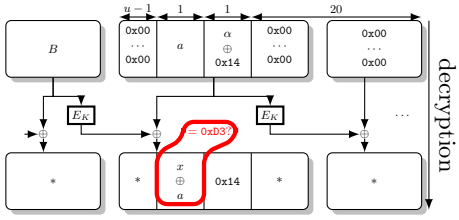


Fig. 4: Recovery of $x = E_K(B)[u]$, knowing $\alpha = E_K(B)[u+1]$.

*Recovering* $E_K(B)[u]$ *knowing* $E_K(B)[u+1]$. We can use the knowledge of $E_K(B)[u+1]$ to speed up the recovery of $E_K(B)[u]$. We fix $b = E_K(B)[u+1] \oplus \texttt{0x14}$ and let byte $a$ take all possible values. This fixes the second byte of the MDC packet header after decryption to the `0x14` value and lets the first byte of the header take all possible $2^8$ values. Thus the format oracle returns `True` for a unique value $a'$, and we have through Equation 1: $E_K(B)[u] = \texttt{0xD3} \oplus a'$. We give a graphical representation of this procedure in Figure 4. The number of requests to recover this byte is $2^8$. It is straightforward to adapt this procedure to recover $E_K(B)[u+1]$ using the knowledge of $E_K(B)[u], 1 \leq u \leq n-1$, for a cost of $2^8$ requests. By incrementally applying these procedures after initially recovering two consecutive bytes of the encrypted value of the block, we can recover $E_K(B)$ with $2^{16} + (n-2)2^8$ requests. Overall, we can decrypt the whole ciphertext with $\|C^*\|(2^{16} + (n-2)2^8)$ requests.

**Improved Recovery Procedure.** It is possible to lower further the complexity of these attacks to $2^8$ requests per byte to decrypt for long messages. These optimisations are detailed in Appendix D.

## 5 Security analysis

**Mitigation.** A basic way of preventing the attacks presented in section 4 would be to remove all leakage through error messages produced by the application during decryption. Adopting this measure for cryptographic libraries and applications that may be used as backend for other applications, ensures that the leakage is not mishandled by calling applications. It can be seen as a misuse-resistance property.

However this may not remove more implicit forms of format oracles. A sound way to prevent format oracle attacks would be to thwart chosen-ciphertext attacks by systematically using authenticated encryption and implementing a thorough "Verify-then-Decrypt" paradigm during decryption, so that no check is performed on the decryption result before the integrity of the ciphertext is cryptographically verified. Such a paradigm cannot be implemented in the case of OpenPGP, since the ciphertext needs to be decrypted before the integrity can be verified. However, it is still possible to obtain a safe implementation by adopting a "Verify-then-Release" paradigm, i.e., the decryption result is buffered, and any processing of the decryption result is deferred until the integrity has been verified. Note that for this solution to be effective, serious compatibility issues can be raised. Indeed, in the case of OpenPGP, this solution would require to deprecate the decryption of messages encrypted without integrity protection. Indeed, if this operation leaks information, it can be used to attack packets encrypted with integrity protection through a downgrade attack, as detailed in Appendix C.

Note also that this cannot be reduced to the choice of the authenticated encryption mode: the implementation of the decryption procedure is crucial to the security of authenticated encryption. In the case of GnuPG implementation of OpenPGP encryption with integrity protection, decryption and MAC computations are performed in parallel, and the decryption result is interpreted on the fly before MAC verification. This behaviour is at odds with the security models under which authenticated encryption modes are evaluated. Even an implementation of the "Encrypt-then-MAC" paradigm, mode that is perceived to be generally safe, can be implemented in this way, and may thus be vulnerable to oracle attacks.

When relying on authenticated encryption to prevent format oracles is not an option, cryptographic toolkits/libraries have to settle for establishing a safe API, that is resistant against misuse by a non cryptography-savvy developer. This can be done by eliminating as much as possible potential leakage of information, by providing a high-level API free of such leakage, and/or explicitly advertising the sensitive information leaked by the API. Errors raised by the exported functions are part of the API and should be considered when studying the leakage channels.

**Future Perspective: State-of-the-Art Authenticated Encryption.** The international competition CAESAR, which aims at identifying good authenticated encryption schemes, has seen the formalisation of security properties related to the security of the decryption procedure, e.g., the definition of the *Release of Unverified Plaintext* setting [3]. Some of the CAESAR candidates may turn out to provide a solution suitable in the context of OpenPGP, and resistance against chosen-ciphertexts attacks even if the decrypted value is processed on-the-fly. Another interesting venue of research that can be followed is to consider the decryption errors into the security model used to study the authenticated encryption mode, following [6]. In both cases however, it is not easily achievable for existing applications to adopt an authenticated encryption mode satisfying stronger security notions. In the case of OpenPGP, it would require an update of the OpenPGP specifications and may entail interoperability issues.

**Disclosure.** We reported our findings to the developers of three OpenPGP applications/libraries: GnuPG, End-to-End and OpenPGP.js.

GnuPG developers acknowledged that unattended usages of their library leaked information via the standard error stream. However, they consider that GnuPG is not at fault, and that it is the responsibility of the users and integrators not to mishandle the leaked warnings and error messages. A patch partially removing the MDC packet header oracle has been integrated in GnuPG 1.4.17 and GnuPG 2.0.23. The inclusion in GnuPG documentation of a security disclaimer warning against padding/format oracles was also discussed.

Google End-to-End developers stated that "the API contract [they] should try to follow is that users of [the high-level API of the library] should be safe to print to an untrusted adversary the errors it throws" and that information leaks conveying useful information should be considered as security bugs. As a result, they tracked every unhandled exceptions in End-to-End. They also removed the MDC packet header leak. OpenPGP.js developers followed a similar approach.

## 6 Conclusion

We highlighted potential format oracles in the OpenPGP message format. If the error messages of the three OpenPGP applications/libraries we studied, GnuPG, OpenPGP.js, and End-to-End, are mishandled, the format oracles are implemented, and they can be used to decrypt data encrypted using OpenPGP.

Modern cryptographic standards should provide strong integrity guarantees, by using only authenticated encryption. Cryptographic application and library providers should be aware of the presented attacks, and develop their products with the following idea in mind: while decrypting a message, *no* information should be leaked about the plaintext until the integrity has been checked. In particular, this includes format constraint checks and timing info leaks. The safest way to meet this requirement is to forbid *any* processing of the plaintext until the integrity has been checked. Library developers should at least strive to specify high-level API free of format oracles, and document the outputs of their libraries that are susceptible to leak sensitive information.

## References


1. Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext Recovery Attacks against SSH. In *IEEE Symposium on Security and Privacy*, pages 16–26. IEEE Computer Society, 2009.
2. Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society, 2013.
3. Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to Securely Release Unverified Plaintext in Authenticated Encryption. Cryptology ePrint Archive, Report 2014/144, 2014. `http://eprint.iacr.org/`.
4. Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012.
5. Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.
6. Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. On Symmetric Encryption with Distinguishable Decryption Failures. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 367–390. Springer, 2013.
7. Tibor Jager and Juraj Somorovsky. How to break XML encryption. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 413–422. ACM, 2011.
8. Kahil Jallad, Jonathan Katz, and Bruce Schneier. Implementation of Chosen-Ciphertext Attacks against PGP and GnuPG. In Agnes Hui Chan and Virgil D. Gligor, editors, *ISC*, volume 2433 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2002.
9. J.Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007.
10. Vlastimil Klíma and Tomas Rosa. Side Channel Attacks on CBC Encrypted Messages in the PKCS#7 Format. Cryptology ePrint Archive, Report 2003/098, 2003. `http://eprint.iacr.org/`.
11. Serge Mister and Robert J. Zuccherato. An Attack on CFB Mode Encryption as Used by OpenPGP. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 82–94. Springer, 2005.
12. Chris J. Mitchell. Error oracle attacks on CBC mode: Is there a future for CBC mode encryption? In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, volume 3650 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2005.
13. B. Möller, T. Duong, and K. Kotowicz. Google Security Advisory: This POODLE Bites: Exploiting The SSL 3.0 Fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.
14. Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-Recovery Attacks Against Datagram TLS. In *NDSS*. The Internet Society, 2012.

15. Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

## A Discussion on OpenPGP Authenticated Encryption Scheme

OpenPGP provides encryption with integrity protection, claiming to use the Hash-then-Encrypt paradigm ([9, section 5.13], "An MDC is intentionally not a MAC"). This paradigm is known not to satisfy state-of-the-art integrity requirements. However, the first random block $R$ that is prepended to the plaintext before encryption and hash computation acts as a MAC session key, for a MAC of the form $H(K||M)$.[5] Thus, OpenPGP actually seems to implement a form of the MAC-then-Encrypt paradigm. The complete security analysis of this mode is not the object of this paper and is left as an open question.

## B Adapting Crafted Ciphertexts to the Format Oracle

The access to the MDC packet header format oracle may be subject to some conditions on the decrypted ciphertext. We found that it was easy to tweak the submitted ciphertext to accommodate the conditions we encountered.

*Guessing $C_2^*[3]$ and $C_2^*[4]$.* The first real plaintext bytes, $C_2^*[3]$ and $C_2^*[4]$, are part of the tag header. In practice, two cases may arise, depending on the preferences included in the recipient's key:

- the data is compressed, which corresponds to a $T_c$ tag, followed by the compression algorithm $T_a$, thus $C_2^*[3] = T_c$ and $C_2^*[4] = T_a$ ;
- the data is simply a litteral, which leads to a $C_2^*[3] = T_\ell$, with the following bytes encoding the litteral length. This length (and therefore $C_2^*[4]$) can be deduced from the overall encrypted packet size.

*Random Prefix Redundancy.* As seen in section 2, prior encryption, a random prefix with basic redundancy is prepended to the payload. The check on the redundancy at the beginning of decryption is meant to get an early detection method that a wrong key is used to decrypt. In case the key is derived from a passphrase, this enables to detect erroneous passphrase inputs. [11] used this check to instantiate a format oracle that led to the ability to decrypt the first 16 bit of any ciphertext block. This redundancy check may still be present and a failure may suppress the "MDC packet header" oracle. In order to ensure that the random prefix redundancy check is always satisfied, it is sufficient to prefix any ciphertext considered in our attacks with the first two blocks of the target ciphertext: instead of submitting $C$, the attacker submits $C_1^*||C_2^*||C$. In the case of tag format oracles, only the first two bytes of $C_2^*$ are used, with the payload ciphertext immediately following.

[5] This MAC is known to be vulnerable to classical extension attacks, but they seem to be irrelevant in the OpenPGP context, since the MAC value is encrypted.

*Compression.* Some implementations, as GnuPG, process in parallel decryption and decompression, and abort decryption if a decompression error is detected. In order to ensure no compression error occurs, the attacker can tweak the submitted ciphertext to trick the implementation into considering the content as a literal packet. This is relatively easy, since it only depends on the third byte of the second block of the encryption input. If the target ciphertext corresponds to a compressed packet, we have $C_2^*[3] \oplus E_K(C_1^*)[3] = T_c$. If we consider the block $C^{\#} = C_2^* \oplus (\texttt{0x00}_2 || T_\ell \oplus T_c || \texttt{0x00}_{n-3})$, the decryption of ciphertexts of the form $C_1^* || C^{\#} || C$ verifies the redundancy of the prefix and the plaintext first packet will be considered a literal packet.

## C Attacking Any Encrypted Packet Types

We show in this section that the malleability of packet tags allows for masquerading a given packet as a packet of a different type. In particular, it allows us to use the MDC packet header oracle to decrypt any symmetrically encrypted packet. The same conclusion holds for the other format oracles: it is thus possible to use a format oracle present in the encryption-only part of a library to attack an encrypted packet with integrity protection.

*Encrypted Packets without Integrity Protection.* It is worthwhile to note that encrypted packets without integrity protection can also be decrypted by using the MDC packet header oracle. Indeed, the target key packet can be used indifferently for encryption with or without integrity protection. Furthermore, the encryption procedure of encrypted packets without integrity protection is also based on CFB mode, with an all-zero IV and a random prefix composed of a random block of $n$ bytes followed by the repetition of the last two bytes of the random block. The variation introduced by the CFB state resynchronization does not modify the attack principle, it only introduces a slight shift in the splitting of the ciphertext into blocks. In order to decrypt $\texttt{EncPacket}_K^E(C^*)$, an encrypted data packet without integrity protection, containing ciphertext $C^*$, it is enough to recover the encryption of the blocks of the truncated ciphertext obtained by removing the first two bytes of $C^*$. The random prefix redundancy constraints and compression constraints can be satisfied in the same manner as for encrypted integrity protected packets.

Note that, considering only the MDC packet header oracle, this leads to the non-intuitive result that the implementation of an authenticated encryption scheme weakens the security of the encryption scheme without integrity.

*Symmetrically Encrypted Session Key Packets.* We give additional details on symmetrically encrypted key packets. The protection of the key stored by these packets relies on a passphrase. The key packet contains the information necessary to derive a key $K_p$ from the passphrase. If the key packet contains an encrypted key, it can be decrypted into $K$ using $K_p$ with the appropriate block cipher in CFB mode, with an all zero IV. Otherwise, $K_p$ is used directly to process data, $K = K_p$.

The attacks can also be applied to the decryption of symmetrically encrypted session key packets, when they contain an encrypted key. Indeed, the same encryption algorithm, CFB with an all-zero IV is used. Furthermore, by removing the encrypted session key from the target encrypted session key packet, one gets a session key packet for the key derived from the passphrase, that can be used to mount the attack. Contrary to the cases of encrypted data packets, no ciphertext blocks corresponding to a plaintext satisfying the random prefix redundancy are available. However, by performing $2^{16}$ requests, with an identical first block, and all possible values for the first two bytes of the second block, such a pair of ciphertext block can be found, if necessary. Attacking session key packets may be preferable to attacking the data packets they protect, since they are usually shorter, and thus their decryption requires less requests.

By the way, the fact that truncating a session key packet gives a valid key packet that can be used to attack the confidentiality of the initial packet is an undesirable property of the symmetrically encrypted session key packet format. It would have been better to build the format of these packets around a key wrap mechanism, providing both confidentiality and integrity of the session key and its metadata.

## D Details of the Improved Recovery Procedure using the MDC Packet Header Oracle

The cost of the basic recovery procedure described in the previous section can be decomposed, for each block, into an expensive first step that recovers initial knowledge on an encrypted block, followed by several cheaper steps that recover the rest of the encrypted block. Starting from the information leaked by the target ciphertext $C^*$, it would be tempting to apply only the cheaper steps, for a cost of $2^8$ requests per byte to decrypt. But a direct approach fails because of the behaviour of decryption at block boundaries. We describe two procedures enabling to recover initial knowledge on $E_K(B)$ for about $2^8$ requests. We then discuss the cost of decrypting $C^*$ as a function of its length.

*Type I procedure.* This procedure enables to test whether $E_K(B)[n] = a \oplus \texttt{0xD3}$ for one request, provided a test block $T$ satisfying $T[n] = a$ and $E_K(T)[1]$ is known. The format of the request ciphertext is $B||T||(E_K(T)[1] \oplus \texttt{0x14})||\texttt{0x00}_{20}$. A graphical representation is given in Figure 5.

*Type II procedure.* This procedure enables to recover $E_K(B)[1]$ for $2^8$ requests, provided $B[n] = a$ and a test block $T$ satisfying $E_K(T)[n] = a \oplus \texttt{0xD3}$. The format of the request ciphertexts is $T||B||b||\texttt{0x00}_{20}$, with $b$ a byte taking all possible values. A graphical representation is given in Figure 5.

*Decryption Strategy.* Note that each of the previous procedures can be applied to any block to recover a first byte of its encrypted value for a cost of $2^8$ once a collection of $2^8$ test blocks, presenting all variations of $a$, has been obtained.

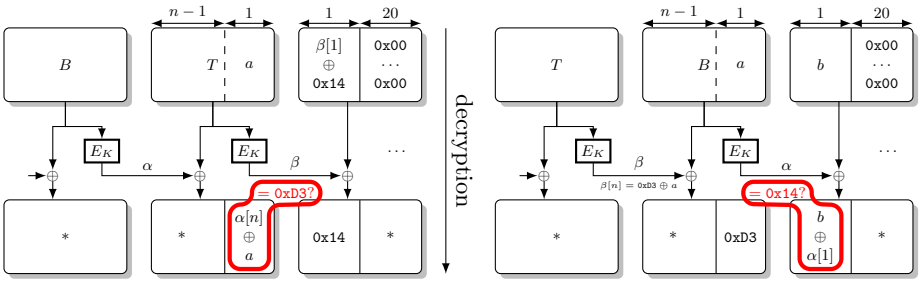


Fig. 5: Type I (left) and type II (right) byte recovery procedures

In order to decrypt the target ciphertext $C^*$, one starts by recovering the block corresponding to the position of the MDC header. Every time a block is recovered, it may provide a new test block that may help to start recovering the remaining ones. Type II and type I procedures are applied as much as possible to start recovering blocks, and the incremental basic procedure is used to finish the encrypted block recovery. If type I and type II procedures cannot be applied, we either apply directly the basic block recovery procedure, or apply type I and type II procedures to extra (pseudo-)ciphertext blocks.

*Decryption Complexity.* We performed simulations to identify the best strategy, with regards to the ciphertext length. If the number of ciphertext block is small, $\|C\| \leq 25$, it is best to apply the basic recovery procedure when the type I and type II procedures cannot be applied. If the number of block is medium $26 \leq \|C\| \leq 267$, the best strategy is to artificially add random ciphertext blocks to have 267 blocks. The message is longer, but it is cheaper to decrypt because the type I and II procedures are applied more often. For long messages, $\|C\| > 267$, it is not necessary to add random blocks, since the cost of decrypting additional block exceeds the cost benefit. In the rare cases were type I and II procedures are not sufficient, one resorts to the basic procedure. Furthermore, for very long messages, the decryption cost is effectively $2^8$ requests per byte. Assuming a 128-bit block cipher, 25 (resp. 267) blocks translate into 400B (resp. 4KB). The results are summarized in Figure 6.

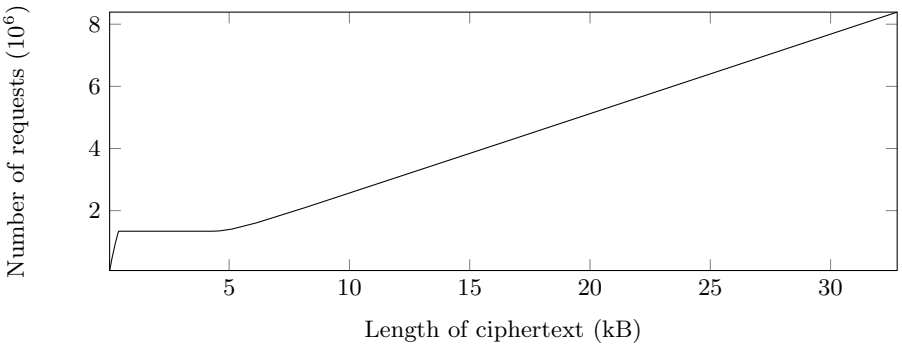


Fig. 6: Number of requests to decrypt a ciphertext as a function of its byte length, assuming 128-bit blocks. When ciphertexts are long enough, the complexity is linear, equal to $2^8$ requests per byte.